Web Request Broker API

Production
March 24, 1996

Contributors

Mala Anand

Matt Bookman

Seshu Adunuthula

Ankur Sharma

For comments and updates contact:

Mala Anand, manand@us.oracle.com

Oracle confidential material

Table of Contents

1 WRB API Overview

The Oracle Web Request Broker (WRB) provides a powerful distributed runtime environment for developing and deploying industrial strength applications for the World Wide Web. The WRB runtime platform enables application developers to write applications that are independent of and work with a number of HTTP Servers.

The Web Request Broker (WRB) API provides a programmatic interface to the Oracle Web Request Broker and enables applications developers to develop their applications (called cartridges in this document) for the WRB. The WRB API provides a higher level API that abstracts away the HTTP protocol while providing the necessary interfaces for application development. In addition, the WRB API provides a rich set of API's for authentication, logging, and attribute management to facilitate application development.

The WRB API and the runtime system together enable web applications to be run on a scalable, distributed high-performance back-end infrastructure. The following document describes the WRB API in detail.

2 WRB API Design Goals & Concepts

The WRB API has been designed to address several deficiencies in the current HTTP Server programmatic interfaces and to meet the following goals:

- The WRB provides a high performance, distributed and scalable application execution framework. It provides applications developers to transparently run multiple instances of their applications and provides the necessary load balancing and system scaling functionality to meet the load on the system. Application developers who develop their applications as WRB cartridges get these advantages of the WRB infrastructure for free.
- The WRB provides a more robust application development environment than NSAPI, ISAPI and other HTTP Server programmatic interfaces:
  - The WRB API provides a higher application level interface for web application developers rather than hooks into the HTTP protocol at various interception points. WRB-enabled applications do not need to aware of the HTTP protocol. The WRB mediates all client requests to the appropriate cartridges and ensures fairness by protecting and preventing cartridges from intercepting requests to other cartridges.
  - WRB applications run in their own address space as independent entities. This provides a level of robustness and reliability not possible with other HTTP Server interfaces which require applications to be dynamically linked into the HTTP Server's address space. This can potentially cause applications to clobber each other's memory space and interfere destructively in signal handlers and other operating system resources. WRB-enabled web applications can be managed and taken on-line and off-line with no affect on other WRB-based web applications running on the same WRB.
  - The WRB provides a high level interface for database access for WRB-enabled applications. This provides the power of the Oracle database for managing configuration and other runtime information in a secure repository, and provides the

application with the infrastructure for maintaining distributed application state and persistance management.

- Any WRB enabled application can invoke other WRB enabled applications as well as make a HTTP request to any other HTTP Server. The WRB provides the distributed infrastructure and dynamically distributes application loads across the network.

- The WRB API provides an easy migration path for current web applications.

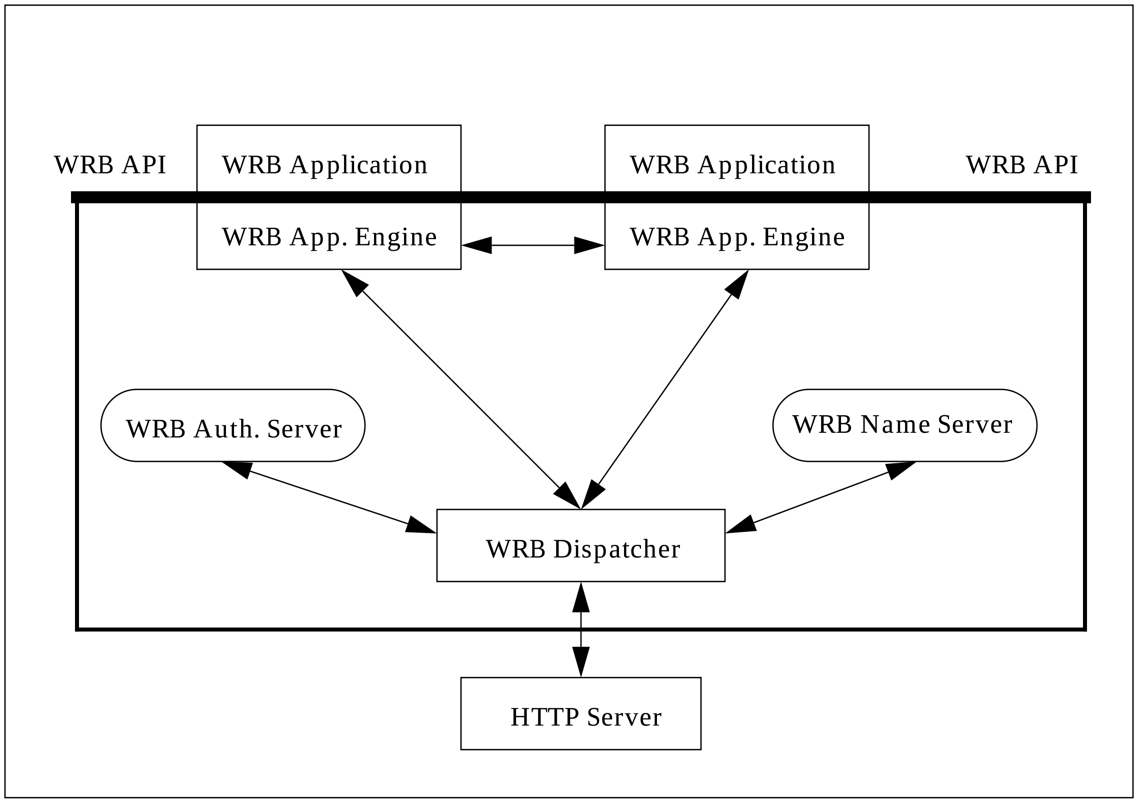

Fig 1: Web Request Broker Architecture

Figure 1 illustrates the Web Request Broker Architecture.

2.1 Web Request Broker Dispatcher

The WRB Dispatcher is the component of the WRB that provides the interaction with different underlying HTTP Servers. It is responsible for routing HTTP requests to the appropriate web application cartridge over the distributed substrate. The HTTP server, in turn, interacts with the web browser using HTTP. The WRB Dispatcher looks up and locates the appropriate cartridge for a request from the WRB configuration and dispatches the request for execution to that cartridge. The WRB can run multiple instances of the web cartridge, and the Dispatcher selects an appropriate instance based on overall system load to deliver optimum application performance. The WRB Dispatcher also authenticates the request against the WRB configuration if the WRB-enabled application is using WRB Authentication Services.

2.2 Web Request Broker Application Engine

Application developers develop their cartridges as shared libraries (on Solaris) and register them with the WRB Dispatcher. The cartridges are loaded in by the WRB Application Engine when a request for that cartridge is received by the WRB Dispatcher. The WRB Application Engine provides the runtime environment for WRB-enabled applications. It enables WRB-enabled applications to receive and send data to the client, as well as the ability to call other cartridges, which may be located on the same WRB or may be distributed across WRB's and other HTTP Servers.

The API's provided by the WRB Application Engine are described below.

3 Application Programming Interface

The WRB API provides application developers with a comprehensive set of calls to build applications independent of the underlying *http* daemons used. The API is organized as follows:

- WRB Application Callbacks
- WRB Listener Information Functions
- WRB Intervention Functions
- WRB Utility Functions

3.1 WRB Application Callbacks

Cartridge developers need to provide the following WRB Application Callbacks that the WRB Application Engine will invoke as described below upon receiving a request from a client for a given cartridge. The bulk of processing by the WRB cartridge is done in the *Exec Callback* where the callback function responds to the listener request and creates a response.

3.1.1 The Init Callback

```
WRBReturnCode WRB_Init( void **clientCtx );
```

This callback is invoked by WRB Application Engine on its initialization. The WRB cartridge initializes its context in this routine which is then made available to the cartridge on subsequent call backs.

3.1.2 The Exec Callback

```
WRBReturnCode WRB_Exec( void *WRBCtx, void *clientCtx );
```

This callback is invoked by WRB Application Engine upon receiving a HTTP Request. The WRB cartridge is responsible for creating the response that is written back to the Listener in this callback .

3.1.3 The Shutdown Callback

```
WRBReturnCode WRB_Shutdown( void *WRBCtx, void *clientCtx);
```

The WRB Application Engine invokes this call back to provide an graceful exit for the WRB Client. Any memory allocated in the client context during the init callback should be deallocated here.

3.1.4 The Reload Callback

```
WRBReturnCode WRB_Reload(void *WRBCtx, void *clientCtx);
```

This callback is not required, but is recommended. The WRB Application Engine invokes this call back whenever the Web Listener has been signalled to reload its own configuration files. The Web Listener halts all new incoming connections, allows existing transactions to complete, and then signals each running WRB cartridge to execute its reload callback (if one exists). If your application uses configuration information from the OWS_APPFILE then you should call WRBGetAppConfig again, as these values may have.

3.1.5 Version Callback

```
char *WRB_Version();
```

The version callback enables the cartridge to return a character string with information about the version of that cartridge. This callback is made by the ““wlctl2” utility

3.1.6 Version Free Callback

```
void WRB_Version_Free();
```

The version free callback enables the cartridge to free the memory allocated by the version callback. This callback is made by the “wlctl2” utility after a successful call to the version callback.

3.2 WRB Listener Information Functions

The following Listener Information functions provide cartridge developers with information, if required, from the client as well as the underlying HTTP Server. All functions return pointers to WRB Application Engine memory which the WRB client should not modify.

3.2.1 WRB Get URL && WRB Get URI

```
char *WRBGetURL( void* WRBCtx );
char *WRBGetURI( void* WRBCtx );
```

Returns the URL/URI received from the HTTP Server. The cartridge developers may extract additional information from the URI/URL if required.

3.2.2 WRB Get Environment Variable

```
char *WRBGetEnvironmentVariable( void *WRBCtx, char *szEnvVar );
```

Returns the value of an environment variable. This provides a way for the WRB client to access CGI Environment variables. Returns NULL if `szEnvVar` points to a invalid environment variable.

3.2.3 WRB Get Environment

```
char **WRBGetEnvironment( void *WRBCtx );
```

Returns the HTTP Server environment as an array of null terminated strings with the syntax `varname=varvalue`. Returns NULL in case of an error.

3.2.4 WRB Get Content

```
char *WRBGetContent( void *WRBCtx );
```

Returns either the Query String or the POST data content depending on the type of the request method. Returns NULL in case on an error.

3.2.5 WRB Get Language

```
char *WRBGetLanguage( void *WRBCtx );
```

Returns the WRB Application Engine Language preferences. Returns NULL in case of an error. On success, returns a comma delimited list of the “Accept” languages.

3.2.6 WRB Get Character Encoding

```
char *WRBGetCharacterEncoding( void *WRBCtx );
```

Returns the WRB Application Engine Character Encoding preferences. Returns NULL in case of an error. On success, returns a comma delimited list of the “Accept” encodings

3.2.7 WRB Get Parsed Content

```
WRBReturnCode WRBGetParsedQueryString(void *WRBCtx, WRBEntry
             **WRBEntries, int *numEntries);
typedef struct {
   char  *name;
   char  *value;
} WRBEntry;
```

Returns an array of name-value pairs after parsing the query string / POST data.

3.2.8 WRB Get Named Entry

```
char *WRBGetNamedEntry( char *entryName, WRBEntry *WRBEntries, int
             numEntries);
```

Returns a name / value pair from a list of entries generated by a call to WRBGetParsedContent. Returns NULL on error

3.2.9 WRB Get ORACLE_HOME

```
char *WRBGetORACLE_HOME( void *WRBCtx );
```

Returns the ORACLE_HOME which was set when the Web Listener was started

3.2.10 WRB Get Application Config

```
char **WRBGetAppConfig( void *WRBCtx );
```

Returns the configuration information for the current application. The structure of the return value is a pointer to an array of pointers to character strings. These character strings are of the form “name=val”. The information is retrieved from the OWS_APPFILE at the time the Web Listener starts up. An example might be:

```
[MyCartidge]
myparam1 = val1
myparam2 = val2
```

3.2.11 WRB Get Application Config Value

```
char *WRBGetConfigVal( void *WRBCtx, char *name );
```

Returns a named item for the application's configuration. The information is retrieved from the OWS_APPFILE at the time the Web Listener starts up.

3.3 WRB Intervention Functions

These functions are invoked by the WRB cartridge to respond to an incoming client request.

3.3.1 WRB Client Read

```
ssize_t WRBClientRead( void *WRBCtx, char *szData, int *nBytes );
```

The WRB Client invokes this function to read the POST data of a HTTP request from the WRB Application Engine. In the current implementation, if a client is going to read POST data it must do so bwfore sending any data (through WRBClientWrite) back to the Web Listener. The return value is the number of bytes read.

3.3.2 WRB Client Write

```
ssize_t WRBClientWrite( void *WRBCtx, char *szData, int nBytes);
```

The WRB cartridge invokes this function to send data to the client. The return value is the number of bytes written.

3.3.3 WRB Client Write File (not implemented)

```
WRBReturnCode WRBClientWriteFile( void *WRBCtx, char *szFileName,
             WRBFileType nFileType );
```

The WRB client invokes this function to send the contents of a file to the client. The file can be located either in the HTTP Servers physical or virtual file system. The file type is specified in the nFileType parameter.

3.3.4 WRB Return HTTP Error

```
ssize_t WRBHTTPReturnError( void *WRBCtx, WRBErrorCode, nErrorCode,
             char *szErrorMesg, boolean close);
```

Invoked by the WRB client when it would like a standard HTTP error sent back to the browser. This call must occur before any other calls to WRBClientWrite, as it is a wrapper which sends : Status <nErrorCode> <szErrorMesg> The "close" flag set to false indicates that you wish to send additional HTTP header information and that you will close the HTTP header explicitly with a call to WRBCloseHTTPHeader.

3.3.5 WRB Client Call Cartridge (not implemented)

```
WRBReturnCode WRBCallCartridge( void *WRBCtx, char *request);
```

The WRB client calls this function to invoke a request which could be satisfied by a cartridge running on the local WRB, or by a remote HTTP Server or WRB.

3.3.6 WRB Client Set Timeout Call (not implemented)

```
WRBReturnCode WRBClientSetTimeOut( void *WRBCtx, int nTimeOut,
             WRBReturnCode (*WRBClientCleanUP)( *void WRBCtx ) );
```

This function can be called by the WRB client to set a time out for long running calls. If WRB Client makes this call the WRB Application Engine sets up a timer to interrupt processing a HTTP request and return to the LIstener with a timedout error response. The WRB client can set a callback `WRBClientCleanUp` to clean up the client in case of a timeout. WRB Application Engine discards any response from the WRB client once the timeout has happened. If no callback is provided a default callback is used.

3.3.7 WRB Return HTTP Redirect

```
ssize_t WRBReturnHTTPRedirect( void *WRBCtx, char *szURI, boolean
             close);
```

Invoked by the WRB client when it would like a standard HTTP error sent back to the browser. This call must come before any other calls to WRBClientWrite, as it is just a wrapper which sends : Status: <nErrorCode> <szErrorMesg>

The “close” flag set to FALSE indicates that you wish to send addition HTTP header information and that you will close the HTTP header explicitly with a call to WRBCloseHTTPHeader.

3.3.8 WRB Close HTTP Header

```
ssize_t WRBCloseHTTPHeaer( void *WRBCtx );
```

Invoked by the WRB client when it would like to close an HTTP header. Generally used after calls to WRBReturnHTTPError or WRBReturnHTTPRedirect with the “close” flag set to FALSE.

3.3.9 WRB Log Message

```
void WRBLogMessage( void *WRBCtx, char *szMessage, int nSeverity );
```

Logs the message `szMessage` to a file `?/ows2/log/wrb_<pid>.c` where ? is the `ORACLE_HOME` and `pid` is the processid of the Web Request Broker. Currently `nSeverity` is not used and reserved for future versions.

4 WRB API Example Application.

The following section describes a simple example of Web Application written using the WRB API functions. This application prints hello world in the browser window

```
#ifndef ORATYPES_ORACLE
# include <oratypes.h>
#endif

#ifndef WRB_ORACLE
# include <wrb.h>
#endif

WRBReturnCode test_init();
WRBReturnCode test_exec();
WRBReturnCode test_shut();
/*****************************************************************/
/* Client-provided callback functions */
/*****************************************************************/
/* The following function defintions should be provided by theWRBclient
/* such that the WRB can invoke the users application.The WRB client */
/* must have a single entry point, which gets specified in the [Apps] */
/* section of the WRB configuration file.                          */
/* */
/* This entry point function should fill in the WRBCallbacks table that
/* it is passed, such that the WRB can call the init, exec, and shut
/* routines respectively (explained below).                        */
/*                                                                 */
/* A simple example of a valid entry function would be:            */
/* WRBReturnCode mydyn_init(WRBCallbacks *WRBcalls)
/* {                                                               */
/*    WRBcalls->init_WRBCallback = my_init;                        */
/*    WRBcalls->exec_WRBCallback = my_exec;                        */
/*    WRBcalls->shut_WRBCallback = my_shut;                        */
/*                                                                 */
/*    return WRB_DONE                                              */
/* }                                                               */

WRBReturnCode testentry (WRBCalls)
WRBCallbacks *WRBCalls;
{
      WRBCalls->init_WRBCallback = test_init;
      WRBCalls->exec_WRBCallback = test_exec;
      WRBCalls->shut_WRBCallback = test_shut;

      return (WRB_DONE);
}
```

```
WRBReturnCode test_init( WRBCtx, clientcxp )
void *WRBCtx;
void **clientcxp;
{
        return (WRB_DONE);
}


WRBReturnCode test_exec( WRBCtx, clientcxp )
void *WRBCtx;
void *clientcxp;
{
      WRBClientWrite(WRBCtx, “Content-type: text/html\n\n
              HelloWorld\n”, 40);
      return (WRB_DONE);
}

WRBReturnCode test_shut( WRBCtx, clientcxp )
void *WRBCtx;
void *clientcxp;
{
      return (WRB_DONE);
}
```

5 Compiling and Linking Cartridges

This section describes the makefile for building Web Request Broker cartridges.

```
#Makefile for building WRB Cartridges
#====================================

LIBHOME    =  $(ORACLE_HOME)/ows2/wrbsdk/lib
INCHOME    =  $(ORACLE_HOME)/ows2/wrbsdk/inc
LDCOM      =  -g -xs -L$(LIBHOME)
SLIBS      =  -lnsl -lm -lsocket -ldl -laio

all: helloworld.so

helloworld.o:helloworld.c
          $(CC)-c -o $@ -g -I$(INCHOME) helloworld.c

#The link line for the final .so dynamic library is given below
helloworld.so: helloworld.o
          $(CC) $(LDCOM) -o $@ -G helloworld.o $(SLIBS)
```

6 Registering Cartridges

6.1 WRB Configuration File specification

All new cartridges will need to register themselves in the WRB config file which is read by the server at startup. The config file has the same name as the listener configuration file but has an extension .app

In the WRB config file the following sections will need to be filled.

6.1.1 [Apps] section

- APPS - This tag will define your cartridge type
- Object Path - This tag will define the full path from where the shared resides.
- Entry Point - This entry point function should fill in the WRBCallbacks table that it is passed, such that the WRB can call the init, exec, and shut routines restively.
- Min - This entry indicates the minimum number of processes that need to be started up for each cartridge/application
- Max - This entry indicates the maximum number of processes that will be allocated for each cartridge/application use.

```
[Apps]
;
; APP   Object Path                         Entry Point  Min     Max
; ===   ===========                         ===========  ===     ===
OWA     /private/oracle/ows2/lib/libndwoa.so ndwoadinit  0       100
SSI     /vobs/ws/src/ssi/ndwussi.so          ndwussinit  0       100
JAVA    /private/oracle/ows2/lib/libjava.so  ojsdinit    0       100
```

6.1.2 [AppsDirs] section

- Virtual Path - this entry specifies the virtual path that all URL's will be refernced by
- APP - Same as above APP entry

- Physical Path - The actual physical path that the applications will read all their data files from.

```
[AppDirs]
;
; Virtual Path          APP     Physical Path
; ============          ===     =============
/ssi                    SSI     /private/oracle/ows2/sample/ssi
/hr/owa                 OWA     /private/oracle/ows2/bin
/tr/owa                 OWA     /private/oracle/ows2/bin
/owa_dba/owa            OWA     /private/oracle/ows2/bin
/java                   JAVA    /private/oracle/ows2/java
```

6.1.3 [AppProtection]

- Virtual Path: The virtual path that needs to be protected. For more information refer to the protection section in listener configuration.
- Protection Scheme: Authentication or Restriction schemes or a combination of both. Refer to the protection section in listener configuration for more information.

```
/owa_dba/owa/*                 Basic(Admin Server)
/hr/owa                        Basic(registered) | IP(oracle)
```

6.1.4 Cartridge configuration section

Each cartrdige can specfify its own configuration information which is availabe to it thru the `WRB_GetAppConfig` call. The configuration information should be in the format A = B.

```
[SSI]
EnableLiveHTML     = TRUE
ParseHTMLExtn      = FALSE
EnableExecTag      = TRUE
ExtensionList    = html shtml lhtml
MaxRequests        = 1
```

Appendix A - References

Table 1: References

SpyGlass Server ADI	*Spyglass Server Reference Manual Version 1.1. http://www.spyglass.com:4040/support*
Internet Server API	*A specification for Writing Internet Server Applications. http://www.microsoft.com/intdev/inttech/isapi.htm*
Netscape Server API	T*he Netscape Server API. http://home.netscape.com*
Open Market FastCGI	*FastCGI Specification. http://www.openmarket.com/fcgi-spec.html*